

# Kickstarting Choreographic Programming

Fabrizio Montesi

University of Southern Denmark  
fmontesi@imada.sdu.dk

**Abstract.** We present an overview of some recent efforts aimed at the development of *Choreographic Programming*, a programming paradigm for the production of concurrent software that is guaranteed to be correct by construction from global descriptions of communication behaviour.

## 1 Introduction

Programming communications among the endpoints in a concurrent system is challenging, because it is notoriously difficult to predict how nontrivial programs executed simultaneously may interact [19]. To mitigate this issue, *choreographies* can be used to give precise specifications of communication behaviour [28,1].

A choreography specifies the expected communications among endpoints from a global viewpoint, in contrast with the standard methodology of giving a separate specification for each endpoint that defines its Input/Output (I/O) actions. As an example, consider the following choreographic specification (whose syntax is derived from the “Alice and Bob” notation from [23]):

Alice  $\rightarrow$  Bob : *book*;    Bob  $\rightarrow$  Alice : *money*

The choreography above describes the behaviour of two endpoints, Alice and Bob. First, Alice sends to Bob a *book*; then, Bob replies to Alice with some *money* as payment. The motivation for using a choreography as specification is that it is always “correct by design”, since it explicitly describes the intended communications in a system. In other words, a choreography can be seen as a formalisation of the communication flow intended by the system designer.

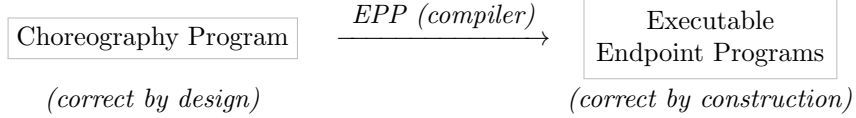
A choreography can be compiled to the local specifications of the I/O actions that each endpoint should perform [27,18,10], as depicted below:

$$\begin{array}{ccc} \boxed{\text{Choreography Spec.}} & \xrightarrow{EPP} & \boxed{\text{Endpoint Spec.}} \\ (correct\ by\ design) & & (correct\ by\ construction) \end{array}$$

In the methodology above, an Endpoint Projection (EPP) procedure is used to generate the specifications for each endpoint starting from a global choreographic specification. The endpoint specifications are therefore correct by construction, because they are computed from a correct-by-design choreography. A major consequent benefit is that such endpoint specifications are also deadlock-free,

because I/O actions cannot be defined separately in choreographies and are therefore always paired correctly in the result of EPP.

In this paper, we give an overview of some recent results by the author and collaborators aimed at applying the choreography-based methodology as a fully-fledged programming paradigm, rather than as a specification method. In this paradigm, called *Choreographic Programming*, choreographies are concrete programs and EPP is a compiler targeting executable distributed code:



Ideally, this methodology will allow developers to program systems from a global viewpoint, which is less error-prone than writing endpoint programs directly, and then to obtain executable code that is correct by construction.

To kickstart the development of choreographic programming, we are interested in finding suitable language models (§ 2) and their implementation (§ 3). We discuss them in the remainder of the paper, following the syntax from [20].

## 2 Language models

In [11] we present the Choreography Calculus (CC), a language model for choreographic programming that follows the correct-by-construction methodology discussed in § 1 and provides an interpretation of concurrent behaviour in choreographies. The key first-class elements of CC are *processes* and *sessions*, respectively representing endpoints that execute concurrently and the conversations among them. The basic statement of choreographic programs, ranged over by  $C$ , is a communication:

$$p.e \rightarrow q.x : k; C$$

which reads “process  $p$  sends the value of expression  $e$  to process  $q$ , which receives it on variable  $x$ , over session  $k$ ; then, the system executes the continuation choreography  $C$ ”. We comment the model by giving the following toy example on a replicated journaling file system.

*Example 1 (Replicated Journaling File System, write operation).* We define a choreography, denoted  $C_{\text{jfs}}$ , in which a client  $c$  uses a session  $k$  to send some data to be written in a journaling file system replicated on two storage nodes.

$$C_{\text{jfs}} \stackrel{\text{def}}{=} \begin{array}{l} 1. \quad c.data \rightarrow j_1.data_1 : k; \\ 2. \quad c.data \rightarrow j_2.data_2 : k; \\ 3. \quad j_1.blocks(data_1) \rightarrow s_1.blocks_{s_1} : k'; \\ 4. \quad j_2.blocks(data_2) \rightarrow s_2.blocks_{s_2} : k'; \\ 5. \quad j_1 \rightarrow c : k; \\ 6. \quad j_2 \rightarrow c : k \end{array}$$

In the choreography  $C_{\text{jfs}}$ , the client  $c$  uses session  $k$  to send the *data* to be written to two processes,  $j_1$  and  $j_2$ , which we assume log the operation in their

respective journals upon reception (Lines 1–2). The two journal processes then use another session,  $k'$ , to forward the data to be written to their respective processes handling the actual data storage,  $s_1$  and  $s_2$  (Lines 3–4). Finally, at the same time, processes  $j_1$  and  $j_2$  send an empty message on session  $k$  to the client, in order to inform it that the operation has been completed (Lines 5–6).

*Concurrency.* Process identifiers ( $c$ ,  $j_1$ ,  $j_2$ ,  $s_1$  and  $s_2$  in our example) are key to formalising concurrent behaviour in CC. Observe Lines 3–4: since processes run in parallel, the communication between  $j_2$  and  $s_2$  in Line 4 could be completed before the communication between  $j_1$  and  $s_1$  in Line 3. In CC, the semantics of the sequential operator is thus relaxed by a syntactic *swapping congruence relation*  $\simeq_C$ , which allows two statements to be swapped if they do not share any processes. For example, the choreography  $C_{jfs}$  would be equivalent to a choreography  $C'_{jfs}$ , denoted  $C_{jfs} \simeq_C C'_{jfs}$ , where in  $C'_{jfs}$  Lines 3 and 4 are exchanged. In [20], the relation  $\simeq_C$  is validated by showing that it corresponds to the typical interleaving semantics of the parallel operator found in process calculi.

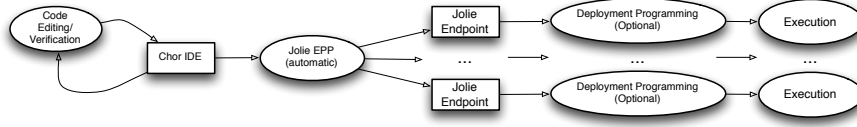
*Sessions and Typing.* The communications in Lines 1–2, 5–6 and the communications in Lines 3–4 are included in different sessions, respectively  $k$  and  $k'$ . Each session represents a logically-separate conversation, as in other session-based calculi (e.g., [15,4]), and is strongly typed in CC with a typing discipline that checks for adherence to protocols expressed as multiparty session types [16]. We give an example of how protocols are mapped to choreographies in § 3.

*Endpoint Projection.* CC comes with an EPP that compiles choreographies to distributed implementations in terms of the  $\pi$ -calculus [11]. The generated code follows that of the originating choreography, according to a small-step operational semantics. As a corollary, the produced code is also deadlock-free: senders and receivers are always ready to communicate when they have to, as I/O actions cannot be mismatched in choreographies.

*Modularity.* In [22], we extend CC to support the implementation and reuse of external libraries/services (modular development), using a notion of external participants in sessions. For example, we can split the choreography  $C_{jfs}$  in two modules, a client choreography  $C_{cli}$  and a server choreography  $C_{srv}$ :

$$\begin{array}{ll}
C_{cli} \stackrel{\text{def}}{=} \begin{array}{l} 1. c.data \rightarrow J1 : k; \\ 2. c.data \rightarrow J2 : k; \\ 3. J1 \rightarrow c : k; \\ 4. J1 \rightarrow c : k \end{array} & C_{srv} \stackrel{\text{def}}{=} \begin{array}{l} 1. C \rightarrow j_1.data_1 : k; \\ 2. C \rightarrow j_2.data_2 : k; \\ 3. j_1.blocks(data_1) \rightarrow s_1.blocks_{s_1} : k'; \\ 4. j_2.blocks(data_2) \rightarrow s_2.blocks_{s_2} : k'; \\ 5. j_1 \rightarrow C : k; \\ 6. j_2 \rightarrow C : k \end{array}
\end{array}$$

The choreographies above refer to each other using references to external processes, e.g.,  $J1$  in  $C_{cli}$  is a reference to process  $j_1$  in  $C_{srv}$ . Separate choreography modules can be compiled and deployed separately, with the guarantee that their generated implementations will interact with each other as expected.



**Fig. 1.** Chor, development methodology (from [11]).

```

program simple;
protocol SimpleProtocol {
  C -> S: hi( string )
}
public a: SimpleProtocol

main
{
  client[C] start server[S] : a( k );
  client.msg -> server.x : wrong( k )
}

```

Operation wrong is not expected by the type for session k  
Press 'F2' for focus

**Fig. 2.** Chor, example of error reporting (from [20]).

*Extraction.* In [12], we present a proofs-as-programs Curry-Howard correspondence between Internal Compositional Choreographies (ICC, a simplification of CC) and a generalisation of Linear Logic [14], inspired by [8]. ICC is a first step in defining a canonical model for choreographies and formalising logical reasoning on choreographic programs. In such correspondence, EPP is formalised as a transformation between logically-equivalent proofs, one corresponding to a choreography program and the other corresponding to a  $\pi$ -calculus term. The transformation is invertible, yielding a procedure for automatically extracting the choreography that a  $\pi$ -calculus term typed with linear logic is following.

### 3 Implementation

The Choreography Calculus (CC), along with related work on models for choreography languages [27,18,10], offers insight on how choreographic programming can be formally understood as a self-standing paradigm. To practically evaluate choreographic programming, we developed the Chor programming language<sup>1</sup>, an open source prototype implementation of CC [20].

In Chor, the correct-by-construction methodology of choreographic programming is proposed as a concrete software development process, depicted in Fig. 1. Choreographies are written using an Integrated Development Environment (IDE), which visualises on-the-fly errors regarding syntax and protocol verification, as in the screenshot in Fig. 2. Then, a choreography can be projected to executable

<sup>1</sup> <http://www.chor-lang.org/>

code via an implementation of EPP that follows the ideas of CC. In this case, the target language is Jolie<sup>2</sup> [21]. Once the compiler has generated the Jolie programs for the endpoints described in the choreography, the developer can customise their deployments. This is done using the Jolie primitives for integrating with standard communication protocols and technologies, which do not alter the behaviour of the code generated by the Chor compiler. The resulting code can finally be executed using the Jolie interpreter.

In Chor, the syntax from CC is extended with operation names for communications (as in Web Services [2]) and data manipulation primitives. As an example, we show an extended implementation of the scenario from Example 2.

---

```

1  protocol Write {
2      C -> J1: { write(string);
3                  C -> J2: write(string);
4                  J1 -> C: ok(void);
5                  J2 -> C: ok(void),
6                  writeAsync(string);
7                  C -> J2: writeAsync(string)
8      }
9  }
10
11 protocol Store { J1 -> S1: write(string);
12                J2 -> S2: write(string) }
13
14 define computeBlocks(j1, j2) { /* ... */ }
15
16 define write(c, j1, j2, s1, s2)
17 (k[ Write:c[C], j1[J1], j2[J2] ],
18  k2[ Store:j1[J1], j2[J2], s1[S1], s2[S2] ]) {
19     if (sync)@c {
20         c.data -> j1.data : write(k);
21         c.data -> j2.data : write(k);
22         computeBlocks( j1, j2 );
23         j1.blocks -> s1.blocks : write( k2 );
24         j2.blocks -> s2.blocks : write( k2 );
25         j1 -> c : ok( k );
26         j2 -> c : ok( k )
27     } else {
28         c.data -> j1.data : writeAsync( k );
29         c.data -> j2.data : writeAsync( k );
30         computeBlocks( j1, j2 );
31         j1.data -> s1.data : write( k2 );
32         j2.data -> s2.data : write( k2 )
33     }
34 }

```

---

We briefly comment the program above, referring the reader to [20] for a more complete description of Chor. Procedure `write` implements the behaviour of the

<sup>2</sup> <http://www.jolie-lang.org/>

processes from Example 2. The sessions  $k$  and  $k2$  ( $k$  and  $k'$  in Example 2) are typed by the protocols `Write` and `Store` respectively. In Line 19, the client  $c$  checks its internal variable `sync` to determine whether the write operation should be synchronous or not. In the first case we proceed as in Example 2. Otherwise, process  $c$  uses a different operation `writeAsync` to notify the others that it does not expect a confirmation message at the end.

## 4 Related Work

The idea of using choreography-like descriptions for communication behaviour has been used for a long time, for example in software engineering [17], security [9,7,5], and specification languages for business processes [28,1].

The development of the formal models that we described in § 2 was made possible by many other previous works on languages for expressing communication behaviour. The notion of session in CC is a variation of that presented in [4] for a process calculus. The theory of modular choreographies was inspired by the article [3], where types for I/O actions are mixed with types for global communications, and by Multiparty Session Types [16], from which we took the type language to interface compatible choreographies. Interestingly, combining multiparty session types with choreographies yields a type inference technique and a deadlock-freedom analysis that do not require additional machinery as in other works in the context of processes [4]. The criteria for a correct Endpoint Projection (EPP) procedure was investigated in many settings, e.g., in [27,6,18,10].

The Chor language and its compiler have already been used as basis for implementing other projects. For example, AIOJ [26] is a choreographic language supporting the update of executable code at runtime, equipped with a formal calculus that ensure deadlock-freedom [25]. Choreographies have also been applied for the design of communication protocols. In particular, Scribble is a specification language for protocols written from a global viewpoint [29], which can be used to generate correct-by-construction runtime monitors (see, e.g., [24]).

## 5 Conclusions and Future Work

We presented some recent efforts aimed at kickstarting the development of choreographic programming as a fully-fledged programming paradigm. While the paradigm holds potential, there is still a lot of work to be done before reaching a productive real-world programming framework. We describe below some possible research directions, some of which are planned for in the current research project behind Chor, the CRC project<sup>3</sup>.

*Integration.* A key factor for the adoption of choreographic programming will be interoperability with existing software. Chor can be extended with local computation primitives that would interact with libraries written in other programming languages, e.g., Java or Scala, similarly to how it is done in Jolie [21].

<sup>3</sup> <http://www.chor-lang.org/>

*Classification.* Just like there are many different language models for different aspects of concurrent programming, e.g., code mobility and multicast, it should be possible to similarly extend choreography models. This suggests a potential benefit in having systematic classifications of choreography languages, to observe the effect that such extensions have on expressiveness and see how far the correct-by-construction methodology can be applied.

*Exceptions.* Introducing exception handling in choreography program raises the issue of coordinating many participants in a global escape (as in [13]), and whether a suitable strategy can always be found, statically or at runtime.

*Formal Implementation.* The EPP procedure in CC is based on  $\pi$ -calculus channels, but its implementation in Chor uses data (protocol headers) to route messages instead, as in many other enterprise frameworks [2]. To the best of the author’s knowledge, realising  $\pi$ -calculus channels using data-based message routing has still to be formally investigated, and the implementation of Chor could provide an initial stepping stone in such a study.

**Acknowledgements.** The author was supported by the Danish Council for Independent Research project *Choreographies for Reliable and efficient Communication software* (CRC), grant no. DFF-4005-00304, and by the EU COST Action IC1201 *Behavioural Types for Reliable Large-Scale Software Systems* (BETTY).

## References

1. Business Process Model and Notation. <http://www.omg.org/spec/BPMN/2.0/>.
2. WS-BPEL OASIS Web Services Business Process Execution Language. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html>.
3. P. Baltazar, L. Caires, V. T. Vasconcelos, and H. T. Vieira. A Type System for Flexible Role Assignment in Multiparty Communicating Systems. In *Proc. of TGC*, 2012.
4. L. Bettini, M. Coppo, L. D’Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
5. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140. IEEE Computer Society, 2009.
6. M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition, 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*, pages 34–50, 2007.
7. S. Briaies and U. Nestmann. A formal semantics for protocol narrations. *Theor. Comput. Sci.*, 389(3):484–511, 2007.
8. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.
9. C. Caleiro, L. Viganò, and D. A. Basin. On the semantics of Alice&Bob specifications of security protocols. *Theor. Comput. Sci.*, 367(1-2):88–122, 2006.

10. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8, 2012.
11. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
12. M. Carbone, F. Montesi, and C. Schürmann. Choreographies, logically. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, pages 47–62, 2014.
13. M. Carbone, N. Yoshida, and K. Honda. Asynchronous session types: Exceptions and multiparty interactions. In *SFM’09*, volume 5569 of *LNCS*, pages 187–212. Springer, 2009.
14. J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
15. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP’98*, volume 1381 of *LNCS*, pages 22–138, Heidelberg, Germany, 1998. Springer-Verlag.
16. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, volume 43(1), pages 273–284. ACM, 2008.
17. International Telecommunication Union. Recommendation Z.120: Message sequence chart, 1996.
18. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Proc. of SEFM*, pages 323–332. IEEE, 2008.
19. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ACM SIGARCH Computer Architecture News*, 36(1):329–339, 2008.
20. F. Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. [http://fabriziomontesi.com/files/m13\\_phdthesis.pdf](http://fabriziomontesi.com/files/m13_phdthesis.pdf).
21. F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. 2014.
22. F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, pages 425–439, 2013.
23. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.
24. R. Neykova, N. Yoshida, and R. Hu. SPY: local verification of global protocols. In A. Legay and S. Bensalem, editors, *RV*, volume 8174 of *LNCS*, pages 358–363. Springer-Verlag, 2013.
25. M. D. Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Deadlock freedom by construction for distributed adaptive applications. *CoRR*, abs/1407.0970, 2014.
26. M. D. Preda, S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli. AIOCJ: A choreographic framework for safe adaptive distributed applications. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, pages 161–170, 2014.
27. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982, United States, 2007. IEEE Computer Society Press.
28. W3C WS-CDL Working Group. Web services choreography description language version 1.0. <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>, 2004.
29. N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble Protocol Language. In *TGC*, pages 22–41, 2013.